Efficient Distributed Detection of Conjunctions of Local Predicates

Michel Hurfin, Masaaki Mizuno, Michel Raynal, Mukesh Singhal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Distributed Detection of Conjunctions of Local Predicates

M. Hurfin M. Mizuno M. Raynal M. Singhal

N° 2731

Novembre 1995

PROGRAMME 1

ISSN 0249-6399

Efficient Distributed Detection of Conjunctions of Local Predicates

M. Hurfin * M. Mizuno ** M. Raynal *** M. Singhal ****

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Adp

Rapport de recherche n ° 2731 — Novembre 1995 — 35 pages

Abstract: Global predicate detection is a fundamental problem in distributed systems and finds applications in many domains such as testing and debugging distributed programs. This paper presents two efficient distributed algorithms to detect conjunctive form global predicates in distributed systems. The algorithms detect the first consistent global state that satisfies the predicate even if the predicate is unstable. The algorithms are based on complementary approaches and are dual of each other. The algorithms are distributed because the predicate detection efforts as well as the necessary information is equally distributed among the processes. We prove the correctness of the algorithms and compare their performance with those of the existing predicate detection algorithms. The proposed algorithms compare very favorably with the existing algorithms in terms of the number of messages exchanged for predicate detection.

Key-words: Distributed systems, On the fly global predicate detection

(Résumé : tsvp)

This work was supported in part * by an INRIA grant (postdoctoral fellowship), ** by the National Science Foundation under Grant CCR-9201645 and Grant INT-9406785, **** by an INRIA grant when the author was visiting IRISA

* IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France, hurfin@irisa.fr
** Dept. of Comp. & Inf. Science, KSU, Manhattan, KS 66506 USA, masaaki@cis.ksu.edu
*** IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr
**** Dept. of Comp. & Inf. Science, OSU, Columbus, OH 43210 USA, singhal@cis.ohio-state.edu

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Des algorithmes répartis performants pour détecter des conjonctions de prédicats locaux

Résumé : Dans les systèmes répartis, la détection de prédicats globaux est un problème fondamental qui trouve des applications dans des domaines tels que le test et la mise au point de programmes répartis. Cet article présente deux algorithmes répartis permettant de détecter efficacement des prédicats globaux exprimés sous forme de conjonction de prédicats locaux. Ces algorithmes permettent de détecter le premier état global cohérent qui satisfait le prédicat même si ce dernier est instable. Les algorithmes sont fondés sur des approches duales et complémentaires. Les algorithmes sont répartis au sens où l'effort de détection en terme de calcul et de stockage d'information est équitablement réparti entre les processus. Nous prouvons ces algorithmes et comparons leur performances avec celles d'autres algorithmes de détection de prédicats. La comparaison avec les algorithmes déjà existants tourne à l'avantage des algorithmes que nous proposons lorsque le critère retenu est le nombre de messages échangés.

Mots-clé : Systèmes répartis, Détection au vol de prédicats globaux.

1 Introduction

Development of distributed applications requires the ability to analyze their behavior at run time whether to debug or control the execution. In particular, it is sometimes essential to know if a property is satisfied (or not) by a distributed computation. Properties of the computation, which specify desired (or undesired) evolutions of the program's execution state, are described by means of predicates over local variables of component processes.

A *basic* predicate refers to the program's execution state at a given time. These predicates are divided into two classes called *local predicates* and *global predicates*. A local predicate is a general boolean expression defined over the local state of a single process, whereas a global predicate is a boolean expression involving variables managed by several processes. Due to the asynchronous nature of a distributed computation, it is impossible for a process to determine the total order in which the events occurred in the physical time. Consequently, it is often impossible to determine the global states through which a distributed computation passed through, complicating the task of ascertaining if a global predicate became true during a computation.

Basic predicates are used as building blocks to form more complex class of predicates such as linked predicates [14], simple sequences [5, 9, 1], interval-constrained sequences [1], regular patterns [4] or atomic sequences [8, 9]. The above class of properties are useful in characterizing the evolution of the program's execution state, and protocols exist for detecting these properties at run time by way of language recognition techniques [2].

When the property (*i.e.*, a combination of the basic properties) contains no global predicate, the detection can be done locally without introducing any delays, without defining a centralized process and without exchanging any control messages. Control information is just piggybacked to the existing message of the application. However, if the property refers at least to one global predicate, then all possible observations of the computation must be considered. In other words, the detection of the property requires the construction and the traversal of the lattice of consistent global states representing all observations of the computation. When the property reduces to one global predicate, the construction of the lattice can be avoided in some cases. If the property is expressed as a disjunction of local predicates, then obviously no cooperation between processes is needed in order to detect the property during a computation. A form of global predicate, namely, the conjunction of local predicates, has been the focus of research [5, 6, 7, 12, 17] during the recent years. In such predicates, the number of global states of interest in the lattice is considerably

reduced because all global states that includes a local state where the local predicate is false need not be examined.

Previous Work

The problem of global predicate detection has attracted considerable attention lately and a number of global predicate detection algorithms have been proposed in the recent past. In the centralized algorithm of Cooper and Marzullo [3], every process reports each of its local states to a process, which builds a lattice of the global computation and checks if a state in the computation satisfies the global predicate. The power of this algorithm lies in generality of the global predicates it can detect; however, the algorithm has a very high overhead. If a computation has n processes and if m is the maximum number of events in any process, then the lattice consists of $O(m^n)$ states in the worst case. Thus, the worst case time complexity of this algorithm is $O(m^n)$. The algorithm in [10] has linear space complexity; however, the worst case time complexity is still linear in the number of states in the lattice.

Since the detection of generalized global predicates by building and searching the entire state space of a computation is utterly prohibitive, researchers have developed faster, more efficient global predicate detection algorithms by restricting themselves to special classes of predicates. For example, a form of global predicate that is expressed as the conjunction of several local predicates has been the focus of research [5, 6, 7, 12, 17] recently. Detection of such predicates can be done during a replay of the computation [12, 17] or during the initial computation [5, 6, 7].This paper focus on the second kind of solution which allows one to detect the predicate even before the end of the computation. In the Garg-Waldecker centralized algorithm to detect such predicates [6], a process gathers information about the local states of the processes, builds only those global states that satisfy the global predicate, and checks if a constructed global state is consistent. In the distributed algorithm of Garg and Chase [7], a token is used that carries information about the latest global consistent state (cut) such that the local predicates hold at all the respective local states. The worst case time complexity of both these algorithms is $O(mn^2)$ which is linear in m and is much smaller than the worst case time complexity of the methods that require searching the entire lattice. However, the price paid is that not all properties can be expressed as the conjunction of local predicates.

Recently, Stoller and Schneider [16] proposed an algorithm that combines the Garg-Waldecker approach [6] with any approach that constructs a lattice to detect Possibly(Φ). (A distributed computation satisfies Possibly(Φ) iff predicate Φ holds in a state in the corresponding lattice.) This algorithm has the best features of both

the approaches – it can detect Possibly(Φ) for any predicate Φ and it detects a global predicate expressed as the conjunction of local predicates in time linear in m (the maximum number of events in any process).

Paper Objectives

This paper presents an efficient distributed algorithm to detect conjunctive form global predicates in distributed systems. We prove the correctness of the algorithm and compare its performance with that of the previous algorithms to detect conjunctive form global predicates.

The rest of the paper is organized as follows: In the next section, we define system model and introduce necessary definitions and notations. Section 3 presents the first global predicate detection algorithm and gives a correctness proof. The second algorithm is presented in Section 4. In Section 5, we compare the performance of the proposed algorithms with the existing algorithms for detecting conjunctive form global predicates. Finally, Section 6 contains the concluding remarks.

2 System Model, Definitions, and Notations

2.1 Distributed Computations

A distributed program consists of n sequential processes denoted by P_1,P_2,$\cdots$,P_n. The concurrent execution of all the processes on a network of processors is called a distributed computation. The processes do not share a global memory or a global clock. Message passing is the only way for processes to communicate with one another. The computation is asynchronous: each process evolves at his own speed and messages are exchanged through communication channels, whose transmission delays are finite but arbitrary. We assume that no messages are altered or spuriously introduced. No assumption is made about the FIFO nature of the channels.

2.2 Events

2.2.1 Definition and Notations

Activity of each process is modeled by a sequence of *events* (*i.e.*, executed action). Three kinds of events are considered: internal, send, and receive events. Let e_i^x denote the x^{th} event which occurs at process P_i. Figure 1 shows an example of distributed computation involving two processes P_1 and P_2. In this example, event e_1^2 is a send event and event e_2^1 is the corresponding receive event. Event e_1^1 is an internal event.

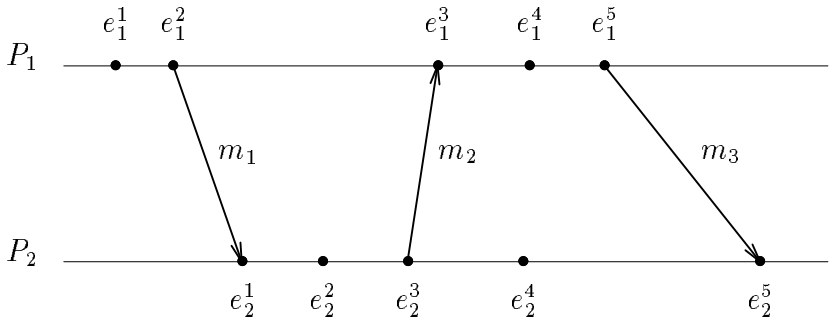

Figure 1: A distributed computation

For each process P_i, we define an additional internal event denoted as e_i^0 that occurred at process P_i at the beginning of the computation. So, during a given computation, execution of process P_i is characterized by a sequence of events:
$\mathrm{E}_i \equiv e_i^0\ e_i^1\ e_i^2 \cdots e_i^x \cdots$.

Furthermore, if the computation terminates, the last action executed at process P_i (denoted as $e_i^{m_i}$) is followed by an imaginary internal event denoted as $e_i^{m_i+1}$.

2.2.2 Causal Precedence Relation Between Events

The "happened-before" causal precedence relation of Lamport induces a partial order on the events of a distributed computation. This transitive relation, denoted by $\prec$, is defined as follows:

$$\forall e_i^x, \forall e_j^y,\ e_i^x \prec e_j^y \iff \begin{cases} (i=j) \wedge (x<y) \\ \text{or} \\ \text{There exists a message } m \text{ such that} \\ \quad e_i^x \text{ is a send event (sending of } m \text{ to } P_j) \text{ and} \\ \quad e_j^y \text{ is a receive event (receiving of } m \text{ from } P_i) \\ \text{or} \\ \text{There exists an event } e_k^z \text{ such that:} \\ \quad e_i^x \prec e_k^z \text{ and } e_k^z \prec e_j^y \end{cases}$$

This relation is extended to a reflexive relation denoted $\preceq$.

2.3 Local states

2.3.1 Definition and Notations

At a given time, the local state of a process P_i is defined by the values of the local variables managed by this process. Although occurrence of an event does not necessarily cause a change of the local state, we identify the local state of a process at a given time with regard to the last occurrence of an event at this process. We use σ_i^x to denote the local state of P_i during the period between event e_i^x and event e_i^{x+1}. The local state σ_i^0 is called the initial state of process P_i.

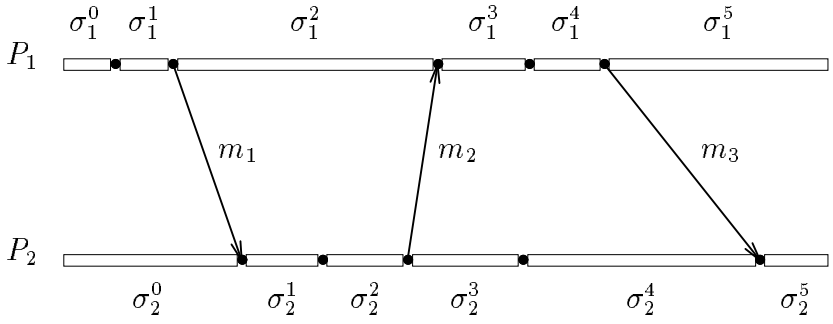

Figure 2: Local states of processes

2.3.2 Causal Precedence Relation Between Local States

The definition of the causal precedence relation between states (denoted by $\longrightarrow$) is based on the happened-before relation between events. This relation is defined as follows:

$$\forall \sigma_i^x,\ \forall \sigma_j^y,\quad \sigma_i^x \longrightarrow \sigma_j^y \iff e_i^{x+1} \preceq e_j^y$$

Two local states σ_i^x and σ_j^y are said to be *concurrent* if there is no causal dependency between them (*i.e.*, $\sigma_i^x \not\longrightarrow \sigma_j^y$ and $\sigma_j^y \not\longrightarrow \sigma_i^x$).

A set of local states is *consistent* if any pair of elements are concurrent. In the distributed computation shown in Figure 2, $\{\sigma_1^0\}$, $\{\sigma_1^1,\sigma_2^0\}$ and $\{\sigma_1^2,\sigma_2^4\}$ are three examples of consistent sets of local states.

2.4 Intervals

2.4.1 Definition and Notations

Since causal relations among local states of different processes are caused by send and receive events, we introduce the notion of *intervals* to identify concurrent sequences of states of a computation. An interval is defined to be a segment of time on a process that begins with a send or receive event (called a communication event) and ends with the next send or receive event. Thus, a process execution can be viewed as a consecutive sequence of intervals.

In order to formally define intervals, we first introduce a new notation to identify communication events. We use ε_i^x to denote the x^{th} send or receive event at P_i. Thus, for each ε_i^x, there exists exactly one e_i^y that denotes the same event. Furthermore, the n imaginary event e_i^0 are renamed as ε_i^0. If the computation terminates, imaginary event $e_i^{m_i+1}$ at process P_i is renamed as $\varepsilon_i^{l_i+1}$ (l_i is the number of communication events that occurred at process P_i).

The x^{th} interval of process P_i, denoted by θ_i^{x-1}, is a segment of the computation that begins at ε_i^{x-1} and ends at ε_i^x. Thus, the first interval at P_i is denoted by θ_i^0. If the computation terminates, the last interval of process P_i is identified by $\theta_i^{l_i}$.

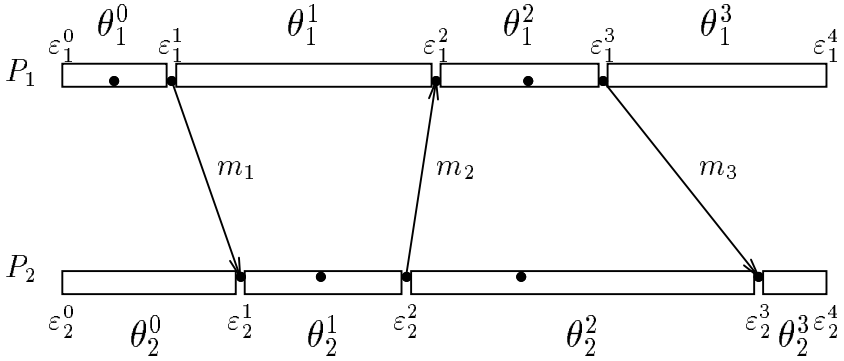

Figure 3: The corresponding set of intervals

We say that interval θ_i^x contains the local state σ_i^y (or σ_i^y is contained in θ_i^x) if the following property holds: $(\varepsilon_i^x \preceq e_i^y) \wedge (e_i^y \prec \varepsilon_i^{x+1})$. This relation is denoted by $\sigma_i^y \in \theta_i^x$. If this relation does not hold, it is denoted by $\sigma_i^y \notin \theta_i^x$. By definition, any interval contains at least one local state.

2.4.2 Causal Precedence Relation Between Intervals

The relation that expresses causal dependencies among intervals is denoted as $\rightarrow$. This relation induces a partial order on the intervals of distributed computation and is defined as follows:

$$\forall \theta_i^x,\ \forall \theta_j^y,\ \ \theta_i^x \ \rightarrow\ \theta_j^y \quad \Longleftrightarrow \quad \varepsilon_i^{x+1} \preceq \varepsilon_j^y$$

A set of intervals is consistent if for any pair of intervals in the set, say θ_i^x and θ_j^y, $\neg(\theta_i^x \rightarrow \theta_j^y)$.

2.5 Global States

A global state (or cut) is a collection of n local states containing exactly one local state from each process P_i. A global state is denoted by $\{\ \sigma_1^{x_1}, \sigma_2^{x_2}, \cdots, \sigma_n^{x_n}\ \}$. If a global state $\{\ \sigma_1^{x_1}, \sigma_2^{x_2}, \cdots, \sigma_n^{x_n}\ \}$ is consistent, it is identified by $\Sigma(x_1,\ x_2,\ \cdots,\ x_n)$.

The set of all consistent global states of a distributed computation form a lattice whose minimal element is the initial global state $\Sigma(0,\ 0,\ \cdots,\ 0)$. An edge exists from $\Sigma(x_1,\ \cdots,\ x_i,\ \cdots,\ x_n)$ to $\Sigma(x_1,\ \cdots,\ x_i+1,\ \cdots,\ x_n)$ if the distributed computation can reach the latter from the former when process P_i executes its next event $e_i^{x_i+1}$. Each path of the lattice starting at the minimal element corresponds to a possible observation of the distributed computation. Each observation is identified by a sequence of events where all events of the computation appear in an order consistent with the "happen before" relation of Lamport. The maximal element $\Sigma(m_1,\ m_2,\ \cdots,\ m_n)$ is called the final global state and exists only if all the processes of the distributed computation have terminated.

Given a computation and a predicate on a global state Φ, we can use the two modal operators proposed by Cooper and Marzullo [3] to obtain two different properties, namely, Possibly(Φ) and Definitely(Φ). A distributed computation satisfies Possibly(Φ) if and only if the lattice has a consistent global state verifying the predicate Φ, whereas Definitely(Φ) is satisfied by the computation if and only if each observation (i.e., each path in the lattice) passes through a consistent global state verifying Φ. In this paper, we focus on the class of global predicates formed as the conjunction of local predicates and we consider only the first satisfaction rule: Possibly(Φ). This rule is particularly attractive to test and debug distributed executions.

2.6 Conjunctions of Local Predicates

A local predicate defined over the local state of process P_i is denoted as $\mathcal{L}_i$. Notation $\sigma_i^x \models \mathcal{L}_i$ indicates that the local predicate $\mathcal{L}_i$ is satisfied when process P_i is in the local state σ_i^x. Due to its definition, a local predicate $\mathcal{L}_i$ can be evaluated by process P_i at any time without communicating with any other process.

We extend the meaning of symbol $\models$ to intervals as follows:

$$\theta_i^x \models \mathcal{L}_i \iff \exists \sigma_i^y \text{ such that } (\sigma_i^y \in \theta_i^x) \wedge (\sigma_i^y \models \mathcal{L}_i)$$

Let Φ denote a conjunction of p local predicates. Without loss of generality, we assume that the p processes involved in the conjunction Φ are $P_1, P_2, \cdots, P_p$. In this paper, we write either Φ or $\mathcal{L}_1 \wedge \mathcal{L}_2 \wedge \cdots \wedge \mathcal{L}_p$ to denote the conjunction.

A set of p local states $\{ \sigma_1^{x_1}, \sigma_2^{x_2}, \cdots, \sigma_p^{x_p} \}$ is called a *solution* if and only if:

$$\begin{cases} \forall i,\ 1 \leq i \leq p,\ \ \sigma_i^{x_i} \models \mathcal{L}_i \\ \wedge \\ \{ \sigma_1^{x_1}, \sigma_2^{x_2}, \cdots, \sigma_p^{x_p} \} \text{ is a consistent set.} \end{cases}$$

A global state $\Sigma(x_1, x_2, \cdots, x_n)$ is called a *complete solution* if this set of local states includes a solution.

By definition, Possibly(Φ) is verified if there exists a complete solution. A consistent set of local states containing less than n local states may be completed to form a consistent global state (*i.e.*, a consistent set of n elements), and thus, a solution may be extensible to one or more complete solutions. The goal of a detection algorithm is not to calculate the whole set of complete solutions but only to determine a solution. This approach is not restrictive. In order to deal with complete solutions rather than with solutions, a programmer can simply add to the conjunction, $n - p$ local predicates $\mathcal{L}_{p+1}, \mathcal{L}_{p+2}, \cdots, \mathcal{L}_n$ which are true in any local state. Consequently, we will no longer speak about complete solution.

Due to the link between local states and intervals, the following definition of a solution is obviously consistent with the first one.
A set of p local states $\{ \sigma_1^{y_1}, \sigma_2^{y_2}, \cdots, \sigma_p^{y_p} \}$ is a *solution* iff there exists a set of intervals $\{\theta_1^{x_1}, \theta_2^{x_2}, \cdots, \theta_p^{x_p}\}$ such that:

$$\begin{cases} \forall i,\ 1 \leq i \leq p,\ \ \sigma_i^{y_i} \models \mathcal{L}_i \\ \wedge \\ \forall i,\ 1 \leq i \leq p,\ \ \sigma_i^{y_i} \in \theta_i^{x_i} \\ \wedge \\ \{ \theta_1^{x_1}, \theta_2^{x_2}, \cdots, \theta_p^{x_p} \} \text{ is a consistent set.} \end{cases}$$

Let $\mathcal{S}$ denote the set of all solutions. If $\mathcal{S}$ is not empty, the *first solution* is the unique element of $\mathcal{S}$ denoted by $\{ \sigma_1^{f_1} , \sigma_2^{f_2} , \cdots , \sigma_p^{f_p} \}$ such that every element $\{ \sigma_1^{x_1} , \sigma_2^{x_2} , \cdots , \sigma_p^{x_p} \}$ of $\mathcal{S}$ satisfies the following property:

$$\forall i,\ 1 \leq i \leq p, \quad f_i \ \leq \ x_i$$

As the property to detect is expressed as a conjunction of local predicates, this particular solution, if it exists, is well defined in the computation. The set of intervals that includes this solution is also well defined. We denote this set of intervals $\{\theta_1^{F_1}, \theta_2^{F_2}, \cdots, \theta_p^{F_p}\}$ and we say that this set of intervals is the first one which verifies Φ.

3 Detection Algorithms for Conjunction of Local Predicates

3.1 Overview

As mentioned in the previous section, Possibly(Φ) is verified by detecting a set of concurrent intervals, each of which verifies its local predicate. We have developed the following two approaches to resolve this problem:

1. In the first approach, processes always keep track of sets of concurrent intervals. For each such set, each process checks whether its interval in the set verifies its local predicate.

2. In the second approach, a process always keeps track of a set of intervals, each of which verifies its local predicate. For each such set, the process checks whether all intervals in the set are concurrent.

Thus, algorithms designed for those complementary approaches are dual of each other. This section described the algorithm corresponding to the first approach in detail, including its correctness proof. The next section describes an algorithm corresponding to the second approach.

3.2 The First Algorithm

3.2.1 Dependency Vectors

To identify a set of p concurrent intervals, the algorithm keeps track of causal dependencies among intervals by using a vector clock mechanism similar to that described

in [13]. Each process P_i $(1 \leq i \leq n)$ maintains an integer vector $D_i[1..p]$, called the *dependency vector*. Since causal relations between two intervals at different processes are created by communication events (and their transitive relation), values in D_i are advanced only when a communication event takes place at P_i. We use D_i^x to denote the value of vector D_i when process P_i is in interval θ_i^x. This value is computed at the time ε_i^x is executed at process P_i.

Each process P_i executes the following protocol:

1. When process P_i is in interval θ_i^0, all the components of vector D_i are zero.
2. When P_i $(1 \leq i \leq p)$ executes a send event, D_i is advanced by setting $D_i[i] := D_i[i] + 1$. The message carries the updated D_i value.
3. When P_i executes a receive event, where the message contains D_m, D_i is advanced by setting $D_i[k] := \max(D_i[k], D_m[k])$ for $1 \leq k \leq p$. Moreover, the dependency vector is advanced by setting $D_i[i] := D_i[i] + 1$ if process P_i belongs to the set of p processes directly implicated in the conjunction (*i.e.*, $i \leq p$).

When a process P_i $(1 \leq i \leq p)$ is in interval θ_i^x, the following properties are observed [15]:

1. $D_i^x[i] = x$ and it represents the number of intervals at P_i that precede interval θ_i^x.
2. $D_i^x[j](j \neq i)$ represents the number of intervals at process P_j that causally precede the interval θ_i^x.
3. The set of intervals $\{\theta_1^{D_i^x[1]}, \theta_2^{D_i^x[2]}, \cdots, \theta_i^{D_i^x[i]}, \cdots, \theta_p^{D_i^x[p]}\}$ is consistent.
4. None of the intervals θ_j^y such that $y < D_i^x[j]$ (*i.e.*, intervals at P_j that causally precede $\theta_j^{D_i^x[j]}$) can be concurrent with θ_i^x. Therefore, none of them can form a set of intervals with θ_i^x that verifies Φ.

Let D_a and D_b be two dependency vector clock values. We use the following notations.

- $D_a = D_b$ iff $\forall i, D_a[i] = D_b[i]$
- $D_a \leq D_b$ iff $\forall i, D_a[i] \leq D_b[i]$

- $D_a < D_b$ iff $(D_a \leq D_b) \wedge \neg(D_a = D_b))$

The following result holds:

- $\theta_i^x \rightarrow \theta_j^y \Leftrightarrow \varepsilon_i^{x+1} \prec \varepsilon_j^y \Leftrightarrow D_j^x < D_i^y$

3.2.2 Logs

Each process P_i $(1 \leq i \leq p)$ maintains a log, denoted by Log_i, that is a queue of vector clock entries. Log_i is used to store the value of the dependency vector associated with the past local intervals at P_i that may be a part of a solution (*i.e.*, intervals that verify $\mathcal{L}_i$ and have not been tested globally yet). Informations about causal relations of a stored interval with intervals at other processes will be used in the future when the stored interval will be examined.

When P_i is in a local state σ_i^y, contained in an interval θ_i^x, such that $\sigma_i^y \models \mathcal{L}_i$, it enqueues vector clock value D_i^x in Log_i if this value has not already been stored. Even if there exists more than one local state in the same interval θ_i^x that verifies $\mathcal{L}_i$, the value D_i^x needs to be logged only once since we are interested in intervals instead of states.

3.2.3 Cuts

In addition to the vector clock, each process P_i $(1 \leq i \leq n)$ maintains an integer vector $C_i[1..p]$, called a *cut* and a boolean vector $B_i[1..p]$. Vector C_i defines the first consistent global state which could verify Φ. In others words, all previous global states dont satisfy Φ. By definition, C denotes a set of p intervals that may be the first solution. If some informations received by P_i show that this set is certainly not a solution, the cut is immediately updated to a new potential solution. At any time, $C_i[j]$ denotes the number of interval of P_j already discarded and indicate the identity $\theta_j^{C_i[j]}$ of the first interval of P_j not yet eliminated. If the conjunction is satisfied during the computation, the cut C will evolve until it denotes the first solution.

Let $C_i^x[j]$ denotes the value of $C_i[j]$ after the communication event ε_i^x has been executed at P_i. The value of C_i remains unchanged in the interval. Each P_i maintains the values C_i in such a way that none of the intervals that precede event $\varepsilon_j^{C_i[j]}$ at P_j can form a set of intervals that verifies Φ. Therefore, each process P_i $(1 \leq i \leq p)$ may discard any values D_i in Log_i such that $D_i[i] < C_i[i]$. Each P_i $(1 \leq i \leq n)$ also maintains the vector B_i in such a way that $B_i[j]$ holds if the interval $\theta_j^{C_i[j]}$ at P_j is certain to verify its local predicate. Thus, if the system is not certain whether the

interval verifies its local predicate, $B_i[j]$ is set to false. To maintain this condition, the cut C and the B vector must be exchanged among processes. When P_i sends a message, it includes vectors C_i, B_i, and D_i in the message.

3.3 Descriptions of the Algorithm

A formal description of the algorithm is given in Section 3.4. The algorithm consists of the following three procedures that are executed at a process P_i:

- A procedure **A** that is executed each time local predicate $\mathcal{L}_i$ associated to P_i $(1 \leq i \leq p)$ becomes true.
- A procedure **B** that is executed when P_i $(1 \leq i \leq n)$ sends a message.
- A procedure **C** that is executed when P_i $(1 \leq i \leq n)$ receives a message.

In addition to vector clock D_i, cut C_i, boolean vector B_i, and log Log_i, each process P_i $(1 \leq i \leq p)$ maintains a boolean variable $not\_logged\_yet_i$. Variable $not\_logged\_yet_i$ is true iff the vector clock value that is associated with the current interval has not been logged in Log_i. This variable helps avoid logging the same vector clock value more than once in Log_i.

A: When the local predicate $\mathcal{L}_i$ becomes true :

Let σ_i^y be the local state that satisfies the local predicate. If θ_i^x is the interval that includes state σ_i^y, then the vector clock value D_i^x, which is associated with interval θ_i^x, is logged in Log_i if it has not been logged yet. To indicate that the vector clock for this interval has already been logged in, process P_i sets variable $not\_logged\_yet_i$ to false.

Furthermore, if the current interval (denoted by θ_i^x or $\theta_i^{D_i^x[i]}$) is also the oldest interval of P_i not yet discarded (denoted by $\theta_i^{C_i^x[i]}$), $B_i^x[i]$ is set to true to indicate that $\theta_i^{C_i^x[i]}$ satisfied $\mathcal{L}_i$.

B: When P_i sends a message:

Since it is the beginning of a new interval, a process P_i $(1 \leq i \leq p)$ advances the vector clock by setting $D_i[i] = D_i[i] + 1$ and resets variable $not\_logged\_yet_i$ to true.

If the log is empty, none of the intervals that precedes the new interval can form a set of intervals that verifies Φ. In particular, this remark holds for the last interval that ends just when the current execution of procedure B (*i.e.*, the sending action) occurs. Consequently, the last interval (and also all the intervals of P_i that causally

precede this one) can be discarded by setting $C_i[i]$ to the current value of $D_i[i]$. (At this step of the computation, $\varepsilon_i^{D_i[i]}$ is the identity of the current send event).

Finally, P_i $(1 \leq i \leq n)$ sends a message along with C_i, B_i, and D_i.

C: When P_i receives a message from P_j that contains D_j, C_j, and B_j:

Based on the definition of vector clocks, it advances D_i and resets variable $not\_logged\_yet_i$ to true. From the definition of a cut, at any process P_k $(1 \leq k \leq p)$, none of the intervals that precedes interval $\theta_k^{C_i[k]}$ or $\theta_k^{C_j[k]}$ can form a set of concurrent intervals that verifies Φ. Thus, C_i is advanced to the componentwise maximum of C_i and C_j. B_i is updated so that it contains more up-to-date of the information in B_i and B_j.

Then, if $i \leq p$, process P_i deletes log values for intervals that precede $\theta_i^{C_i[i]}$ since these intervals do not belong to sets of concurrent intervals that verify Φ.

After this operation, there are two possibilities:

- **Case 1** Log_i becomes empty *i.e.* Log_i does not contain any interval that occurs after $\theta_i^{C_i[i]}$ and before $\theta_i^{D_i[i]}$: In this case, none of the intervals at P_i represented by $\theta_i^{y_i}$ such that $y_i < D_i[i]$ can form a set of concurrent intervals that verify Φ. The algorithm needs to consider only future intervals, denoted by θ_i^z, such that $D_i[i] \leq z$.

  Since none of the intervals $\theta_k^{y_k}$ such that $y_k < D_i[k]$ at other processes P_k can form a set of concurrent intervals with such future intervals θ_i^z of P_i, cut C_i is advanced to D_i. When process P_i executes the receive action, it has no informations about intervals $\theta_k^{D_i[k]}$ $(1 \leq k \leq p)$. Therefore, all components of vector B_i are set to false.

- **Case 2** Log_i contains at least one entry that was logged after the occurrence of event $\varepsilon_i^{C_i[i]}$: Let the oldest such logged entry be D_i^{log}. From the properties of vector D and the definition of a cut, at any process P_k, $1 \leq k \leq p$, none of the intervals preceding $\theta_k^{D_i^{log}[k]}$ or $\theta_k^{C_i[k]}$ can form a set of concurrent intervals with $\theta_i^{D_i^{log}[i]}$ that verifies Φ. Thus, C_i is advanced to the componentwise maximum of D_i^{log} and C_i. Similar to Case 1, if the value $C_i[k]$ is modified (*i.e.*, it takes its value from $D_i^{log}[k]$), P_i is not certain whether P_k's local predicate held in the interval $\theta_k^{D_i^{log}[k]}$. Thus, $B_i[k]$ is set to false. If the value $C_i[k]$ remains unchanged, the value $B_i[k]$ will also remain unchanged.

Furthermore, since $\theta_i^{D_i^{log}[i]}$ verified its local predicate, $B_i[i]$ is set to true. At this point, P_i checks whether $B_i[k]$ is true for all k. If so, this indicates that each interval in the concurrent set of intervals[1] $\{\theta_1^{C_i[1]}, \theta_2^{C_i[2]}, \cdots, \theta_p^{C_i[p]}\}$ verifies its local predicate and thus, Φ is verified.

3.4 A Formal Description of the Algorithm

Initialization procedure executed by any process P_i

D_i := (0,0,$\cdots$,0); C_i := (0,0,$\cdots$,0); B_i := (false,false,$\cdots$,false);
if $(i \leq p)$ **then**
$Create(Log_i)$; $not\_yet\_logged_i$:= true;
endif

Procedure A executed by process P_i $(1 \leq i \leq p)$
when the local predicate $\mathcal{L}_i$ becomes true

if $(not\_yet\_logged_i)$ **then**
$Enqueue(Log_i, D_i)$; $not\_yet\_logged_i$:= false;
if $(C_i[i] = D_i[i])$ **then** $B_i[i]$:= true; **endif**
endif

Procedure B executed by any process P_i
when it sends a message

if $(i \leq p)$ **then**
$D_i[i] := D_i[i] + 1$; $not\_yet\_logged_i$:= true;
if $Empty(Log_i)$ **then** $C_i[i] := D_i[i]$ **endif**
endif
Append vectors D_i, C_i, and B_i to the message;
Send the message;

Procedure C, executed by any process P_i
when it receives a message from P_j

[1] We will prove in subsection 3.7 that a set of intervals numbered by C_i values always are concurrent.

```
        Extract vectors D_j, C_j and B_j from the message;
        D_i := max(D_i,D_j);
        (C_i,B_i) := Combine_Maxima((C_i,B_i),(C_j,B_j));
        if (i ≤ p) then
          D_i[i] := D_i[i] + 1;   not_yet_logged_i := true;
          while ((not (Empty(Log_i))) and (Head(Log_i)[i] < C_i[i])) do
                Dequeue(Log_i);
                /* Delete all those logged intervals that from the current
                /* knowledge do not lie on a solution.
          if (Empty(Log_i)) then
            C_i := D_i;   B_i := (false,false,···,false);
            /* Construct a solution that passes through the next local interval.
            else
            (C_i,B_i) := Combine_Maxima((C_i,B_i),(Head(Log_i),(false,false,···,false)))
            B_i[i] := true;
            /* Construct a solution that passes through the logged interval.
            if (B_i = (true,true,···,true)) then "Possibly(Φ) is verified" endif
          endif
        endif
        Deliver the message;

    Function Combine_Maxima ((C1,B1), (C2,B2))
        B: vector [1..p] of boolean;
        C: vector [1..p] of integers;
        for j := 1 to p do
            case
              C1[j] > C2[j]   −→   C[j] := C1[j];   B[j] := B1[j];
              C1[j] = C2[j]   −→   C[j] := C1[j];   B[j] := (B1[j] or B2[j]);
              C1[j] < C2[j]   −→   C[j] := C2[j];   B[j] := B2[j];
            endcase
        return(C,B);
```

3.5 A Simple Example

Since the algorithm is quite involved, we illustrate the operation of the algorithm with the help of an example. In Figure 4, a local state contained in an interval is

represented by a grey area if it satisfies the associated local predicate. At different step of the computation, we indicate the values of the main variables used to detect Possibly($\mathcal{L}_1 \wedge \mathcal{L}_2$). Two items with square brackets next to a process interval, respectively, depict the contents of vectors D and C. Those values remain unchanged during the entire interval. The value of vector B after execution of a communication event is indicated between round brackets.

Initial value of interval number at two processes is 0 and C vector is (0 0) at both processes. When the local predicate holds in interval θ_1^0 ($\sigma_1^1 \models \mathcal{L}_1$), process P_1 enqueues D_1 vector into Log_1. Process P_1 also set $B_1[1]$ to true because it is certain that $\theta_1^{C_1[1]}$ satisfied $\mathcal{L}_1$. When P_1 sends message $m1$, it increments $D_1[1]$ to 1 and sends vectors B_1, C_1, and D_1 in the message.

When P_2 receives message $m1$, it increments $D_2[2]$ to 1 and then updates its B, C, and D vectors. P_2 finds its Log empty and constructs a potential solution using its D vector and stores it into its C vector. When the local predicate becomes true in state σ_2^1, P_2 adds D_2 to Log_2. As the variable $not\_yet\_logged_2$ is false when process P_2 is in local state σ_2^2, the vector clock D is not logged twice during the same interval. When P_2 sends message $m2$, it increments $D_2[2]$ to 2 and sends vectors B_2, C_2, and D_2 in the message.

When P_1 receives message $m2$, it increments $D_1[1]$ to 2 and then updates its B, C, and D vectors. After merging with the vectors received in the message, P_1 finds that $C_1[1]$ (=1)>$Head(Log_1)[1]$ (=0) and discards this entry from Log_1. Since Log_1 is empty, P_1 constructs a potential solution using its D vector and stores it into its C vector. When the local predicate becomes true in interval σ_1^2, P_1 logs vector D_1 to Log_1. When P_1 sends message $m3$, it increments $D_1[1]$ to 3 and sends vectors B_1, C_1, and D_1 in the message.

In the meantime, local predicate holds in state σ_2^3 and consequently, P_2 logs vector D_2 to Log_2.

When P_2 receives message $m3$, it increments $D_2[2]$ to 3 and then updates its B, C, and D vectors. After merging with the vectors received in the message, P_2 finds that $C_2[2]$ (=2)>$Head(Log_2)[2]$ (=1) and discards this entry. Since the next entry in Log_2 cannot be discarded, P_2 constructs a potential solution using $Head(Log_2)[2]$ vector and stores it into its C and B vectors. The potential solution goes through interval θ_1^2. The fact that this interval satisfies $\mathcal{L}_1$ is known by process P_2 ($B_2[1]$ is true). After P_2 sets $B_2[2]$ to $true$, it finds that all entries of vector B_2 are $true$ and declares the verification of the global predicate.

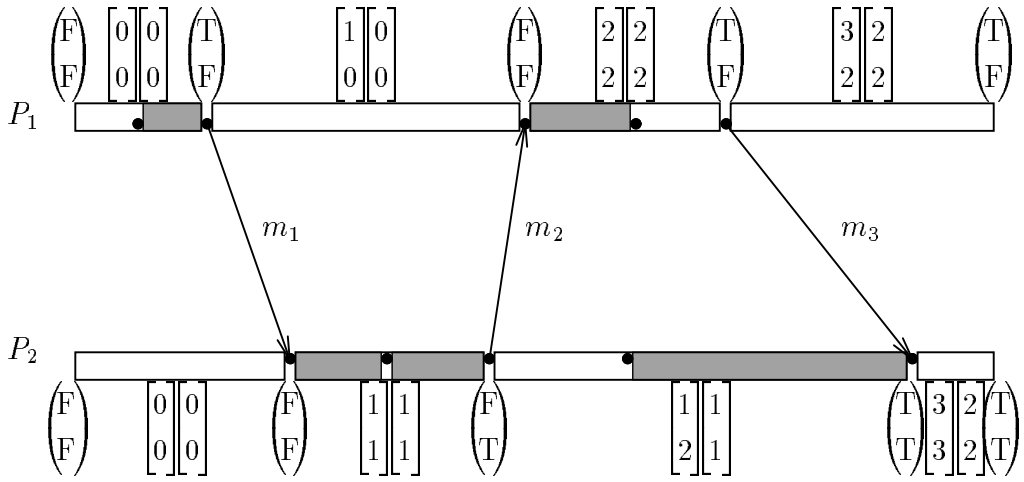

Figure 4: An Example to Illustrate Algorithm 1.

3.6 Extra messages

The algorithm is able to detect if a solution exists without adding extra-message during the computation and without defining a centralized process. The algorithm depends on the exchange of computation message between processes to detect the predicate. As a consequence, not only the detection may be delayed, but also in some cases the computation may terminate and the existing solution may go undetected. For example, if the first solution is the set consisting of the p last intervals, $\{\theta_1^{l_1}, \cdots, \theta_i^{l_i}, \cdots, \theta_p^{l_p}\}$, the algorithm will not detect it. To solve this problem, if a solution has not been found when the computation terminates, messages containing vector D, C, and B are exchanged between the p processes until the first solution is found. To guarantee the existence of at least one solution, we assume that the set of intervals $\{\theta_1^{l_1+1}, \cdots, \theta_i^{l_i+1}, \cdots, \theta_p^{l_p+1}\}$ is always a solution. Extra messages are exchanged only after the computation ends and the first solution has not been detected yet. To reduce the overhead due to these extra messages between processes, one can use a privilege (token) owned by only one process at a time. The token circulates around a ring consisting of the p processes, disseminating the information about the three vectors. Another solution consists of sending the token to a process who may know the relevant information (*i.e.*, a process P_j such that $B_i[j]$ is false).

3.7 Correctness of the algorithm

Lemma 1: Let V_1 and V_2 be two vector timestamps such that the sets of intervals represented by $\{\theta_1^{V_1[1]}, \theta_2^{V_1[2]}, \cdots, \theta_p^{V_1[p]}\}$ and $\{\theta_1^{V_2[1]}, \theta_2^{V_2[2]}, \cdots, \theta_p^{V_2[p]}\}$ are both concurrent. Then, the set of intervals represented by $\{\theta_1^{V_3[1]}, \theta_2^{V_3[2]}, \cdots, \theta_p^{V_3[p]}\}$ is concurrent, where $V_3[i] = \max(V_1[i], V_2[i])$ for $1 \leq i \leq p$,

Proof: We show that a pair of intervals $(\theta_i^{V_3[i]}, \theta_j^{V_3[j]})$ for any combination of i and j, $1 \leq i, j \leq p$ and $i \neq j$, is concurrent. Renumbering the vectors V_1 and V_2 if necessary, suppose $V_3[i] = V_1[i]$. There are two cases to consider:

1. $V_3[j] = V_1[j]$: This case is obvious because, from the assumption, $\theta_i^{V_1[i]}$ and $\theta_j^{V_1[j]}$ are concurrent.

2. $V_3[j] = V_2[j]$: Suppose on the contrary that $\theta_i^{V_3[i]}$ and $\theta_j^{V_3[j]}$ are not concurrent. There are two cases to consider:

   (a) $\theta_j^{V_3[j]} \rightarrow \theta_i^{V_3[i]}$ $(\equiv \theta_j^{V_2[j]} \rightarrow \theta_i^{V_1[i]})$: In this case, $\varepsilon_j^{V_2[j]+1} \prec \varepsilon_i^{V_1[i]}$. Since $V_1[j] \leq V_2[j]$, this implies $\varepsilon_j^{V_1[j]+1} \prec \varepsilon_i^{V_1[i]}$ and so $\theta_j^{V_1[j]} \rightarrow \theta_i^{V_1[i]}$. This contradicts the assumption that $\theta_j^{V_1[j]}$ and $\theta_i^{V_1[i]}$ are concurrent.

   (b) $\theta_i^{V_3[i]} \rightarrow \theta_j^{V_3[j]}$ $(\equiv \theta_i^{V_1[i]} \rightarrow \theta_j^{V_2[j]})$: By applying the same argument as the case (a), this leads to a contradiction to the assumption that $\theta_i^{V_2[i]}$ and $\theta_j^{V_2[j]}$ are concurrent.

Thus, $\theta_i^{V_3[i]}$ and $\theta_j^{V_3[j]}$ are concurrent. □

The following lemma guarantees that a cut C_i always keeps track of a set of concurrent intervals.

Lemma 2: At any process P_i, at any given time, $\{\theta_1^{C_i[1]}, \theta_2^{C_i[2]}, \cdots, \theta_p^{C_i[p]}\}$ is a set of concurrent intervals.

Proof: C_i is updated only in one of the following three ways:

1. When a receive event is executed, by executing $C := D$.

2. When a receive event is executed by taking maximum of C_i and C_j (the cut contained in the message sent by process P_j), and then by taking maximum of C_i and D_i^{log} (The oldest value of the dependency vector still in the log).

3. When a send event occurs, entry $C_i[i]$ is set to $D_i[i]$.

Let $update(C_i^x)$ denote the update on cut C_i at communication event ε_i^x giving the value C_i^x. We define a partial order relation (represented by "$\rightsquigarrow$") on all updates on all cuts C_i $(1 \leq i \leq p)$ as follows:

1. If ε_i^x and ε_i^y are two consecutive communication events that occur at P_i (*i.e.*, $x + 1 = y$), then $update(C_i^x) \rightsquigarrow update(C_i^y)$.

2. If there exists a message m such that ε_i^x is the sending of m to P_j and ε_i^y is the receiving of m from P_i, then $update(C_i^x) \rightsquigarrow update(C_j^y)$.

Let SEQ be a topological sort of the partial order of update events. We prove the lemma by induction on the number of updates in SEQ.

Induction Base: Since the set of intervals $\{\theta_1^0, \theta_2^0, \cdots, \theta_p^0\}$ is concurrent, initially the lemma holds for any C_i (*i.e.*, after $update(C_i^0)$) .

Induction Hypothesis: Assume that the lemma holds up to t applications of the updates.

Induction Steps: Suppose $(t+1)^{st}$ update occurs at process P_i and let C_i^x denote the cut value after the $(t+1)^{st}$ update operation denoted as $update(C_i^x)$.

Case 1: ε_i^x is a receive event and $C_i^x = D_i^x$.

From the definition of vector clocks, for any vector clock value D, the set of intervals $\{\theta_1^{D[1]}, \theta_2^{D[2]}, \cdots, \theta_p^{D[p]}\}$ is concurrent. Thus, C_i^x represents a set of concurrent intervals.

Case 2: ε_i^x is a receive event and $C_i^x = max(max(C_i^{x-1}, C_j^y), D_i^{log})$.

As ε_j^y is the corresponding send event, $update(C_j^y) \rightsquigarrow update(C_i^x)$. Clearly, the relation $update(C_i^{x-1}) \rightsquigarrow update(C_i^x)$ also holds. Since from induction hypothesis, both C_i^{x-1} and C_j^y represent sets of concurrent intervals, $max(C_i^{x-1}, C_j^y)$ represents a set of concurrent intervals (from Lemma 1). Since from the definition of vector clocks, D_i^{log} (the oldest entry still in the log) represents a set of concurrent intervals, C_i^x represents a set of concurrent intervals (from Lemma 1).

Case 3: ε_i^x is a send event, $C_i^x[i] = D_i^x[i]$, and $\forall j$ such that $j \neq i$, $C_i^x[j] = C_i^{x-1}[j]$.

Suppose no receive event occurs at process P_i before the send event ε_i^x. All the entries of vector D_i^x and vector C_i^x, except $D_i^x[i]$ and $C_i^x[i]$, are zero. Therefore, $C_i^x = D_i^x$ and the proof is the same as in Case 1.

Suppose now that ε_i^y is the last receive event who occurs before the sending event ε_i^x. No entry in the log has been discarded since this event occurred. As the Log is empty, we can conclude that the log was also empty when this receive event occurred and hence, $C_i^y = D_i^y$. As this event is the last receive event that occurs before ε_i^x, we conclude that $\forall j$ such that $j \neq i$, $D_i^y[j] = D_i^x[j]$ and $C_i^y[j] = C_i^x[j]$. Therefore $C_i^x = D_i^x$. Then, the proof is the same as in case 1.

□

The following lemma shows that if there is a solution, a cut C_i will not miss and pass beyond the solution.

Lemma 3: Consider a particular cut (identified by an integer vector S) such that $\{\theta_1^{S[1]}, \cdots, \theta_p^{S[p]}\}$ is a set of concurrent intervals that verifies Φ.

Let ε_i^x denote any communication event. If, for all communication events ε_j^y such that $\varepsilon_j^y \prec \varepsilon_i^x$, $C_i^y \leq S$, then $C_i^x \leq S$.

Proof: Proof is by contradiction. Suppose there exists a communication event ε_i^x such that $\neg(C_i^x \leq S)$, and for any communication event ε_j^y such that $\varepsilon_j^y \rightarrow \varepsilon_i^x$, $C_j^y \leq S$. That is, ε_i^x is the first event that advances C_i beyond S.

There are two cases to consider:

1. $C_i^x[i] \leq S[i]$ and $S[j] < C_i^x[j]$ for some $j \neq i$:

   From hypothesis, $C_i^{x-1}[j] \leq S[j]$ holds. Therefore, entry $C_i[j]$ is modified during execution of event ε_i^x. This event is necessarily a receive event (ε_k^z is the corresponding send event).

   Case 1 : $C_i^x = D_i^x$.

   Note that $\theta_i^{C_i^x[i]}$, $\theta_i^{D_i^x[i]}$ and θ_i^x denote the same interval. $S[j] < D_i^x[j]$ holds. So from the definition of dependency vectors, $\theta_j^{S[j]} \rightarrow \theta_i^x$. However, either ($\theta_i^{C_i^x[i]} = \theta_i^{S[i]}$) or ($\theta_i^{C_i^x[i]} \rightarrow \theta_i^{S[i]}$) and therefore, $\theta_j^{S[j]} \rightarrow \theta_i^{S[i]}$. This contradicts the hypothesis that $\{\theta_1^{S[1]}, \cdots, \theta_p^{S[p]}\}$ is a set of concurrent intervals.

Case 2 : $C_i^x = max(max(C_i^{x-1}, C_k^z), D_i^{log})$.

From the assumption, $C_i^{x-1} \leq S$ and $C_k^z \leq S$. Therefore, $C_i^x[j] = D_i^{log}[j]$. $S[j] < D_i^{log}[j]$ holds. So from the definition of dependency vectors, $\theta_j^{S[j]} \rightarrow \theta_i^{D_i^{log}[j]}$. Because of the *max* operation, $D_i^{log}[i] \leq C_i^x[i]$ and so $\theta_j^{S[j]} \rightarrow \theta_i^{C_i^x[i]}$. Then the proof is the same as in Case 1.

2. $C_i^x[i] > S[i]$:

Case 1 : $C_i^x[i] = D_i^x[i]$

ε_i^x is either a send or a receive event.

From the algorithm, it is clear that this case occurs only if none of the intervals that occurred between $\theta_i^{C_i^{x-1}[i]}$ (including this) and $\theta_i^{D_i^x[i]}$ verifies $\mathcal{L}_i$. This contradicts the fact that $\theta_i^{S[i]}$ verifies $\mathcal{L}_i$ since $C_i^{x-1}[i] \leq S[i] < C_i^x[i]$ $(= D_i^x[i])$.

Case 2 : $C_i^x[i] = max(max(C_i^{x-1}[i], C_k^z[i]), D_i^{log}[i])$. ε_i^x is a receive event and ε_k^z is the corresponding send event. From the assumption, $C_i^{x-1}[i] \leq S[i]$ and $C_k^z[i] \leq S[i]$. Assume that $C_i^\chi = max(C_i^{x-1}[i], C_k^z[i])$. Then $C_i^\chi[i] \leq S[i]$ and therefore $C_i^x[i] = D_i^{log}[i]$.

From the algorithm, it is clear that this case occurs only if none of the intervals that occurred between $\theta_i^{C_i^\chi[i]}$ (including this) and $\theta_i^{D_i^{log}[i]}$ verifies $\mathcal{L}_i$. This contradicts the fact that $\theta_i^{S[i]}$ verifies $\mathcal{L}_i$ since $C_i^\chi[i] \leq S[i] < C_i^x[i]$ $(= D_i^{log}[i])$.

□

The following lemma proves that the algorithm keeps making progress if it has not encountered a solution.

Lemma 4: Suppose process P_i has executed the algorithm at the x^{th} communication event ε_i^x (*i.e.*, P_i is in the interval θ_i^x) and that the set $\{\theta_1^{C_i^x[1]}, \cdots, \theta_i^{C_i^x[i]}, \cdots, \theta_p^{C_i^x[p]}\}$ does not verify Φ. Then, there exists ε_j^y such that $C_i^x < C_j^y$.

Proof: There are two reasons for $\{\theta_1^{C_i^x[1]}, \cdots, \theta_i^{C_i^x[i]}, \cdots, \theta_p^{C_i^x[p]}\}$ not verifying Φ:

1. $\theta_i^{C_i^x[i]}$ does not verify $\mathcal{L}_i$:

In this case, at ε_i^x, P_i could not find an interval that verifies $\mathcal{L}_i$, and therefore, $C_i^x[i]$ was set to x (*i.e.*, the value of $D_i^x[i]$). At the next communication event ε_i^{x+1}, P_i updates at least the i^{th} entry of C_i by setting $C_i^{x+1}[i]$ to $D_i^{x+1}[i]$. Thus $C_i^x < C_i^{x+1}$.

2. There exists at least one process P_k $(1 \leq k \leq p)$ such that $\theta_k^{C_i^x[k]}$ does not verify $\mathcal{L}_k$:

   In this case, P_k will eventually advance $C_k[k]$ to a value greater than $C_i^x[k]$ (refer to Case 1). This new value computed when event ε_k^z occurs will propagate to other processes. Extra messages eventually exchanged at the end of the computation guarantee that there will eventually be a communication event ε_j^y at a process P_j such that that $\varepsilon_k^z \prec \varepsilon_j^y$ and $C_i^x < C_j^y$.

□

Finally, the following theorem shows that Φ is verified in a computation iff the algorithm detects a solution.

Theorem: [1] If there exists an interval θ_i^x on a P_i such that, during this interval, $B_i[k]$ holds for all k $(1 \leq k \leq p)$, then $\{\theta_1^{C_i^x[1]}, \cdots, \theta_p^{C_i^x[p]}\}$ verifies Φ.
[2] Conversely, if $\{\theta_1^{C_i^x[1]}, \cdots, \theta_p^{C_i^x[p]}\}$ verifies Φ for an event ε_i^x on some processor P_i, then there exists a communication event ε_j^y such that for all k $(1 \leq k \leq p)$, $C_j^y[k] = C_i^x[k]$ and $B_j[k]$ holds.

Proof:
[1] Proof is by contradiction. Suppose $\theta_k^{C_i^x[k]}$ does not verify $\mathcal{L}_k$ for some k. We show that as long as $C_i[k]$ is not changed, $B_i[k]$ is false. There are two cases to consider:

1. $i = k$ (*i.e.*, $\theta_i^{C_i^x[i]}$ does not verify $\mathcal{L}_i$):

   $C_i[i]$ is updated to $D_i[i]$ when communication events ε_i^x occurs. If ε_i^x is a receive event, $B_i^x[i]$ is set to false at the same time and remains false during the interval θ_i^x. Log_i is necessarily empty for the entire duration of interval $\theta_i^{C_i^x[i]}$. $C_i[i]$ is modified only when event ε_i^{x+1} occurs.

   If ε_i^x is a send event, the value of $B_i[i]$ is unchanged since the last receive event or since the beginning of the computation if no receive event occurs at process P_i before the send event ε_i^x. In both cases, Log_i remains empty during this entire period and $B_i[i]$ remains false.

2. $i \neq k$:

   P_i is updated $C_i[k]$ to $C_i^x[k]$ because there existed a process P_j that advanced $C_j[k]$ to $C_i^x[k]$, and the value was propagated to P_i. P_j must have set $B_j[k]$ to false and this information must have propagated to P_i. This value was propagated to P_i without going through P_k (else P_k would have been advanced $C_k[k]$ to a value greater than $C_i^x[k]$). It is easy to see that $B_i^x[k]$ is false: Since P_k is the only process that can change $B_k[k]$ to true, P_i will never see $B_i[k] =$ true together with $C_i[k] = C_i^x[k]$.

[2] Assume that $\{\theta_1^{C_i^x[1]}, \cdots, \theta_p^{C_i^x[p]}\}$ verifies Φ. Message exchanges guarantee that there will eventually be a communication event ε_j^y such that for all k, $1 \leq k \leq p$, $\varepsilon_k^{C_i^x[k]+1} \prec \varepsilon_j^y$. When process P_k is in interval $\theta_k^{C_i^x[k]}$, $B_k[k]$ is set to true.

From Lemma 3, once a process P_h sets $C_h[k]$ to $C_i^x[k]$, it does not change this value in the future. This implies that all the processes P_h that are on the path of the message exchange from $\varepsilon_k^{C_i^x[k]+1}$ to ε_j^y, sets $C_h[k]$ to $C_i^x[k]$ and $B_h[k]$ to true — none of such processes P_h sets $B_h[k]$ to false by advancing $C_h[k]$ beyond $C_i^x[k]$. Thus, all information is eventually propagated to P_j, and so $B_j^y[k]$ holds for all k, $1 \leq k \leq p$. □

4 The Second Algorithm

In the second algorithm, every process always keeps track of a set of intervals for all the processes such that each of the intervals verifies its local predicate. For each such set, the process checks whether all the intervals in the set are concurrent.

4.1 Overview of the Algorithm

4.1.1 Verified Intervals

In this algorithm, only the intervals that verify their associated local predicates are of interest. We call such intervals *verified intervals*. A new notation Ω_i^{x-1} is used to identify the x^{th} verified interval of process P_i. Thus, for each Ω_i^x, there exists exactly one θ_i^y that denotes the same interval.

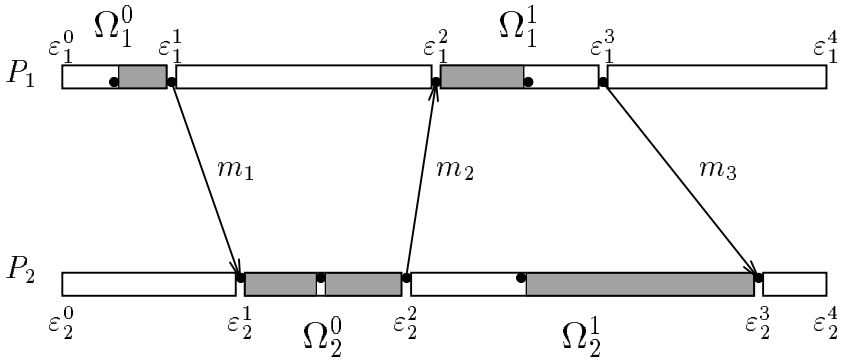

Figure 5: The corresponding set of verified intervals.

4.1.2 Dependency Vectors

As in the first algorithm, each process P_i $(1 \leq i \leq n)$ maintains a dependency vector $D_i[1..p]$. $D_i[i]$ keeps track of the identity of the next verified interval that P_i will encounter. Even though P_i does not know which interval it will be, it knows that the next verified interval will be denoted by $\Omega_i^{D_i[i]}$.

When P_i $(1 \leq i \leq p)$ has encountered the x^{th} verified interval denoted, by Ω_i^{x-1}, it enqueues D_i in Log_i. (A detailed description of Log_i is given later.) At this moment, $D_i[i] = x - 1$. Then it increments $D_i[i]$ by one to x to look for the next verified interval, Ω_i^x. Note that the existence of the verified interval, Ω_i^x, is not guaranteed at this moment. The local predicate may not be satisfied anymore during the computation.

In the first algorithm, vector D remains the same for the entire duration of an interval. In the second algorithm, on the contrary, vector D may change once during an interval if this interval is a verified interval.

In order to capture causal relation among verified intervals at different processes, the following protocol is executed on D_i by a process P_i $(1 \leq i \leq n)$:

1. Initially, all the components of vector D_i are zero.
2. When P_i executes a send event, it sends D_i along with the message.
3. When P_i executes a receive event, where a message contains D_m, D_i is advanced by setting $D_i[k] := \max(D_i[k], D_m[k])$ for $1 \leq k \leq p$.

Clearly, the following properties hold:

1. $D_i[i]$ represents the number of verified intervals at P_i whose existence can be confirmed by P_i (*i.e.*, P_i has already passed through its $(D_i[i])^{st}$ verified interval).

2. $D_i[j](j \neq i)$ represents the number of verified intervals that have occurred at process P_j and causally precede the current interval of P_i.

3. The set of verified intervals $\{\Omega_1^{D_i[1]}, \Omega_2^{D_i[2]}, \cdots, \Omega_i^{D_i[i]}, \cdots, \Omega_p^{D_i[p]}\}$ is not necessarily consistent. Yet, by definition, if interval $\Omega_k^{D_i[k]}$ exists, it satisfies the associated local predicate.

4. None of the intervals Ω_j^y such that $y < D_i[j]$ (*i.e.,* verified intervals at P_j that causally precede $\Omega_j^{D_i[j]}$) can be concurrent with $\Omega_i^{D_i[i]}$. Therefore, none of them can form a set of intervals with $\Omega_i^{D_i[i]}$ that verifies Φ.

4.1.3 Logs

Each process P_i maintains a log, denoted by Log_i, in the same manner as in the first algorithm. When P_i verifies its local predicate $\mathcal{L}_i$, it enqueues the current D_i before incrementing $D_i[i]$ by one.

When Log_i is not empty, notation D_i^{log} is used to denote the value of the vector clock at the head of Log_i. Necessarily, when the log is not empty, the existence of $\Omega_i^{D_i^{log}[i]}$ has already been confirmed by P_i.

4.1.4 Cuts

Like the first algorithm, each process P_i maintains an integer vector C_i and a boolean vector B_i. The meaning of C_i is similar to that of the first algorithm; that is, $\Omega_j^{C_i[j]}$ is the next possible interval at P_j that may be in a solution and none of the verified intervals that precedes $\Omega_j^{C_i[j]}$ can be in a solution. Therefore, each process P_i may discard any values D_i in Log_i such that $D_i[i] < C_i[i]$.

The meaning of B_i is also similar to that in the Algorithm 1. Each P_i maintains B_i in such a way that $B_i[j]$ is true if process P_i is certain that verified interval $\Omega_j^{C_i[j]}$ has been confirmed by P_j. Furthermore, process P_i is certain that none of the verified interval $\Omega_k^{C_i[k]}$ $(1 \leq k \leq p)$ causally precede $\Omega_j^{C_i[j]}$. Thus, if P_i is not certain

whether the verified interval $\Omega_j^{C_i[j]}$ has already been confirmed by P_j, $B_i[j]$ is set to false.

4.2 Descriptions of the Algorithm

A formal description of the algorithm is given in Section 4.3. As the first algorithm, the second algorithm consists of three procedures that are executed at a process P_i. Again, we assume that the set of intervals $\{\theta_1^{l_1+1}, \cdots, \theta_i^{l_i+1}, \cdots, \theta_p^{l_p+1}\}$ is a solution. Extra messages are exchanged after the computation ends only if the first solution has not been discovered yet.

When the local predicate $\mathcal{L}_i$ becomes true:
Let σ_i^y be a local state that satisfies the local predicate. P_i has entered the verified interval Ω_i^x that includes state σ_i^y. It logs D_i in Log_i if D_i has not been logged yet since the beginning of this interval. In order to indicate that the vector clock for this interval has already been logged, it sets variable $not\_logged\_yet_i$ to false. The counter of verified interval $D_i[i]$ is incremented by one to reflect that the current interval is a verified interval.

Furthermore, if the current verified interval is also the oldest verified interval of P_i not yet discarded (denoted by $\Omega_i^{C_i[i]}$), $B_i[i]$ is set to true to confirm the existence of $\Omega_i^{C_i[i]}$.

When P_i sends a message:
Since it marks the beginning of a new interval, P_i resets variable $not\_logged\_yet_i$ to true and then it sends the message along with C_i, B_i, and D_i.

When P_i receives a message from P_j that contains D_j, C_j, and B_j:
Since a new interval begins, it resets variable $not\_logged\_yet_i$ to true. As in the first algorithm, none of the intervals at any process P_k that precede $\Omega_k^{C_i[k]}$ or $\Omega_k^{C_j[k]}$ can form a set of concurrent intervals that verifies Φ. Thus, C_i is advanced to the componentwise maximum of C_i and C_j.

At this moment, B_i is also updated. "$B_i[k]$ is true" means that P_i is *certain* that the existence of $\Omega_k^{C_i[k]}$ has been confirmed. Thus, if $C_i[k] = C_j[k]$ and at least one of $B_i[k]$ and $B_j[k]$ is true, $B_i[k]$ is set to true.

P_i then deletes all entries from Log_i that precede $\Omega_i^{C_i[i]}$ since all those verified intervals are no more potential components of a solution.

After this operation, there are two cases to consider:

- **Case 1** Log_i becomes empty: In this case, none of the verified intervals at P_i up to this moment forms a set of concurrent verified intervals.

  The algorithm needs to consider only verified intervals that will occur in the future. If such intervals exist, $\Omega_i^{D_i[i]}$ will be the first one. Since all of the verified intervals Ω_k^y such that $y < D_i[k]$ at other processes P_k causally precede $\Omega_i^{D_i[i]}$, none of such intervals can be in a solution. Thus, cut C_i is advanced to D_i.

  When process P_i executes the receive action, it is not certain whether P_k ($1 \leq k \leq p$) has encountered the verified interval $\Omega_k^{D_i[k]}$. Therefore, all components of vector B_i are set to false.

- **Case 2** Log_i contains at least one logged interval: Let the oldest of such logged entry be D_i^{log}. From the properties of vector D and the definition of a cut, all of the verified intervals at any process P_k preceding $\Omega_k^{C_i[k]}$ or $\Omega_k^{D_i^{log}[k]}$, causally precede $\Omega_i^{D_i^{log}[i]}$, and none of such intervals can be in a solution. Thus, C_i is advanced to the componentwise maximum of D_i^{log} and C_i.

  Similar to Case 1, if the value $C_i[k]$ is modified (*i.e.*, it takes its value from $D_i^{log}[k]$), P_i is not certain whether P_k's will encounter the verified interval $\theta_k^{D_i^{log}[k]}$. Thus, $B_i[k]$ is set to false. If $C_i[k]$ remains unchanged, $B_i[k]$ also remains unchanged to follow other processes' decision.

  Furthermore, $B_i[i]$ is set to true since P_i has confirmed the existence of $\Omega_i^{D_i^{log}[i]}$.

  At this moment, with the new information (C_j, B_j, and D_j), P_i may be able to detect a solution. Thus, P_i checks whether $B_i[k]$ is true for all k. If so, this indicates that all the verified intervals in set $\{\Omega_1^{C_i[1]}, \Omega_2^{C_i[2]}, \cdots, \Omega_p^{C_i[p]}\}$ have been confirmed and concurrent with one another since no process has detected causal relations between any pair of intervals in the set.

4.3 Formal Description of the Algorithm

Initialization procedure executed by any process P_i

D_i := (0,0,$\cdots$,0); C_i := (0,0,$\cdots$,0); B_i := (false,false,$\cdots$,false);
if ($i \leq p$) **then**
$Create(Log_i)$; $not\_yet\_logged_i$:= true;
endif

Procedure A executed by process P_i $(1 \leq i \leq p)$
when the local predicate $\mathcal{L}_i$ **becomes true**
  if $(not\_yet\_logged_i)$ **then**
    $Enqueue(Log_i,D_i)$; $not\_yet\_logged_i$:= false;
    if $(C_i[i] = D_i[i])$ **then** $B_i[i]$:= true; **endif**
    $D_i[i] := D_i[i] + 1$;
  endif

Procedure B executed by any process P_i
when it sends a message
  if $(i \leq p)$ **then** $not\_yet\_logged_i$:= true; **endif**
  Append vectors D_i, C_i, and B_i to the message;
  Send the message;

Procedure C executed by any process P_i
when it receives a message from P_j
    Extract vectors D_j, C_j, and B_j from the message;
    $D_i := max(D_i,D_j)$;
    $(C_i,B_i) := Combine\_Maxima((C_i,B_i),(C_j,B_j))$;
    if $(i \leq p)$ **then**
      $not\_yet\_logged_i$:= true;
      while ((**not** $(Empty(Log_i))$) **and** $(Head(Log_i)[i] < C_i[i])$) **do**
          $Dequeue(Log_i)$;
          /* Delete all those logged intervals that from the current
          /* knowledge do not lie on a solution.
      if $(Empty(Log_i))$ **then**
        $C_i := D_i$; B_i := (false,false,$\cdots$,false);
        /* Construct a solution that passes through the next local verified interval.
        else
        $(C_i,B_i) := Combine\_Maxima((C_i,B_i),(Head(Log_i),$(false,false,$\cdots$,false)))
        $B_i[i]$:= true;
        /* Construct a solution that passes through the logged verified interval.
        if $(B_i$ = (true,true,$\cdots$,true)) **then** "Possibly(Φ) is verified" **endif**
      endif
    endif
    Deliver the message;

4.4 Discussion

4.4.1 An Example

To help the readers understand the algorithm, in Figure 5, we illustrate the operation of the second algorithm for a computation similar to that one used in Figure 4. In Figure 5, the contents of vector D and C that are indicated next to a process interval in square brackets, are the values of the vectors just after evaluation of the last local state of the interval (*i.e.*, just before execution of the communication event).

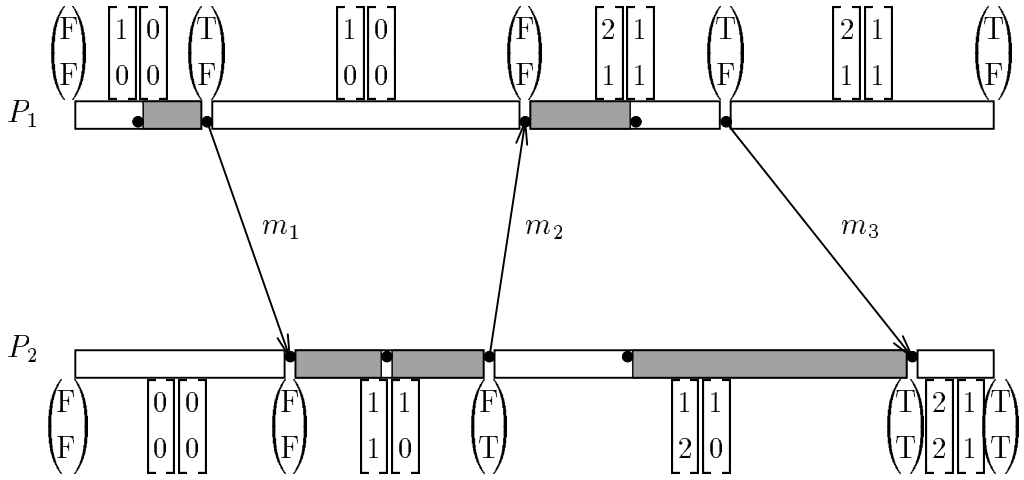

Figure 6: An Example to Illustrate Algorithm 2.

4.4.2 Difference Between Both Approaches

The second algorithm can be considered an optimization of the first one. Interval counters D and C evolve more slowly in the second algorithm and updates of both vectors occur less often. For example, vector C is not modified on a send action. Each algorithm finds the first solution in a different way. In the first algorithm, each interval of the solution is located via a number of communication events that occur before the process encounters this interval. In the second algorithm, the delivered information is the number of validated interval that precede the solution.

The difference between both algorithms is much more on the semantics and the properties of the control variables rather than on the way they are updated. For example, update of vector C is made in a similar way in both algorithms. Yet, each

component is managed as a counter of interval (in the first algorithm) or as a counter of verified interval (in the second algorithm). Both algorithms employ complementary approaches to find the first solution. In the first algorithm, the corresponding set of interval is always concurrent (*i.e.*, it satisfies the first criterion of the solution). In the second algorithm, the elements of the set are always verified intervals (*i.e.*, the set satisfies the second criterion of the solution).

A correctness proof of the second algorithm is similar to the proof of the first algorithm. However, Lemma 1 and Lemma 2 become irrelevant in the second algorithm. Instead, the following lemma becomes useful:

Lemma 5: At any time during execution of P_i, if $B_i[j]$ $(1 \leq j \leq p)$ is true then

$$\forall k, 1 \leq k \leq p, \neg(\Omega_k^{C_i[k]} \rightarrow \Omega_j^{C_i[j]})$$

Proof: Process P_j has necessarily updated $B_j[j]$ to true to account for the fact that interval $\Omega_j^{C_i[j]}$ has been encountered and is the oldest interval not discarded yet. At the same time, P_j used the value at the head of the Log, D_j^{log}, to update all the other components $C_j[k]$ to a value greater or equal to $D_j^{log}[k]$ in order to invalidate all the verified intervals that are in the causal past of $\Omega_j^{C_i[j]}$. Therefore, as the cut C never decreases (due to the merge operation made at the beginning of each receive event), P_i cannot share with P_j the same vision of the values of $B[j]$ and $C[j]$ and at the same time keep an older value for some component $C_i[k]$. □

The rest of the proof is the same as that of the first algorithm with the definition of interval appropriately modified.

5 A Comparison with Existing Work

Previous work in detecting conjunctive form global predicates has been mainly by Garg and Waldecker [6] and Garg and Chase [7]. Garg-Waldecker algorithm is centralized [6], where each process reports all its local states satisfying its local predicate to a checker process. The checker process gathers this information, builds only those global states that satisfy the global predicate, and checks if a constructed global state is consistent. This algorithm has a message, storage, and computation complexities of $O(Mp^2)$ where M is the number of messages sent by any process and p is the number of processes over which the global predicate is defined.

In [7], Garg and Chase present two distributed algorithms for detection of conjunctive form predicates. In these algorithm, all processes participate in the global predicate detection on equal basis. The first distributed algorithm requires vector clocks and employs a token that carries information about the latest global consistent cut such that the local predicates hold at all the respective local states. The message, storage, and computation complexities of this algorithm is the same as of Garg-Waldecker [6] algorithm, namely, $O(Mp^2)$. However, the worst case message, storage, and computation complexities for a process in this algorithm is $O(Mp)$; thus, the distribution of work is more equitable than in the centralized algorithm. The second distributed algorithm does not use vector clocks and uses direct dependencies instead. The message, storage, and computation complexities of this algorithm are $O(Mn)$ and the worst case message, storage, and computation complexities for a process in this algorithm are $O(M)$; thus, this algorithm is desirable when p^2 is greater than n.

The proposed predicate detection algorithm does not cause transfer of any additional messages (except in the end provided the predicate is not detected when the computation terminates). The control information needed for predicate detection is piggybacked on computation messages. On the contrary, the distributed algorithms of Garg and Chase may require exchange of as many as Mp and Mn control messages, respectively. Although the worst case volume of control information exchanged is identical, namely, $O(Mp^2)$, in the first Garg and Chase algorithm and in the proposed algorithm, the latter results in no or few additional message exchanges. A study by Lazowska *et al.* [11] showed that message send and receive overhead can be considerable (due to context switching and execution of multiple communication protocol layers) and it is desirable to send few bigger from performance point of view.

6 Concluding Remarks

Global predicate detection is a fundamental problem in the design, coding, testing and debugging, and implementation of distributed programs. In addition, it finds applications in many other domains in distributed systems such as deadlock detection and termination detection.

This paper presented two efficient distributed algorithms to detect conjunctive form global predicates in distributed systems. The algorithms detect the first consistent global state that satisfies the predicate and work even if the predicate is uns-

table. The algorithms are based on complementary approaches and the second algorithm can be considered an optimization of the first one, where the vectors D and C increase at a lower rate. We proved the correctness of the algorithms. The algorithms are distributed because the predicate detection efforts as well as the necessary information are equally distributed among the processes. Unlike previous algorithms to detect conjunctive form global predicates, the algorithms do not require transfer of any additional messages during the normal computation; instead, they piggyback the control information on computation messages. Additional messages are exchanged only if the predicate remains undetected when the computation terminates.

References

[1] Ö. Babaoğlu and M. Raynal. Specification and Verification of Dynamic Properties in Distributed Computations. *Journal of Parallel and Distributed Computing*, 28(2), pp. 173–185, August 1995.

[2] Ö. Babaoğlu, E. Fromentin, and M. Raynal. Debugging Distributed Executions by Using Language Recognition. In *Proc. of the 24*th *Int. Conf. on Parallel Processing*, Volume II (Software track), pp. 55–62, Wisconsin, August 1995.

[3] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 163-173, Santa Cruz, California, May 1991.

[4] E. Fromentin, M. Raynal, V.K. Garg, and A.I. Tomlinson On the fly testing of regular patterns in distributed computations In *Proc of the the* 23^{rd} *International Conference on Parallel Processing*, pp. 73–76, St. Charles, IL, August 1994.

[5] V.K. Garg and B. Waldecker. Detection of Unstable Predicates in Distributed Programs. In *Proc. of the 12*th *Int. Conf. on Foundations of Software Technology and Theoretical Computer Science*, Springer Verlag, LNCS 652, pp. 253–264, New Delhi, India, December 1992.

[6] V.K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. In *IEEE Transactions on Parallel and Distributed Systems*, 5(3), pp. 299–307, March 1994.

[7] V.K. Garg and C.M. Chase. Distributed Algorithms for Detecting Conjunctive Predicates. In *Proc. of the 15*th *Int. Conf. on Distributed Computing Systems*, pp. 423-430, Vancouver, Canada, June 1995.

[8] D. Haban and W. Weigel, Global Events and Global Breakpoints in Distributed Systems. In *Proc. of the 21*st *Hawaii Int. Conf. on System Sciences* , pp. 166–175, January 1988.

[9] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting Atomic Sequences of Predicates in Distributed Computations. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 32–42, San Diego, California, May 1993 (Reprinted in SIGPLAN Notices, Vol. 28,12, December 1993).

[10] R. Jegou, R. Medina, and L. Nourine. Linear Space Algorithm for On-line Detection of Global States. In *Proc. of the Int. Workshop on Structures in Concurrency Theory*, (to appear), 1995.

[11] E.D. Lazowska, J. Zahorjan, D. Cheriton, and W. Zwaenepoel. File Access Performance of Diskless Workstations, *ACM Transactions on Computer Systems*, 4(3), pp. 238–268, August 1986.

[12] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, volume 15, pp. 62–69, 1992.

[13] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. of the Int. Workshop on Parallel and Distributed Algorithms*, North-Holland, pp. 215-226, Chateau de Bonas, France, October 1988.

[14] B.P. Miller and J.-D. Choi. Breakpoints and Halting in Distributed Programs. In *Proc. of the 8*th *IEEE Int. Conf. on Distributed Computing Systems*, pp. 316–323, San Jose, California, June 1988.

[15] M. Raynal and M. Singhal, Logical Clocks: A Way to Capture Causality in Distributed Systems. To appear in *IEEE Computer*.

[16] S.D. Stoller and F.B. Schneider. Faster Possibility Detection by Combining Two Approaches. In *Proc. of the 9*th *Int. Workshop on Distributed Algorithms*, Springer Verlag, LNCS 972, pp 318–332, Le Mont-Saint-Michel, France, September 1995.

[17] S. Venkatesan and B. Dathan. Testing and Debugging Distributed Programs Using Global Predicates, In *IEEE Transactions on Software Engineering*, 21(2), pp. 163-177, February 1995.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399